LISP NOTES #1

LISP

Acronymed from List Processing, or from Lots of Irritating Silly Parentheses ;)
It was developed by John MacCarthy and his group in late 1950s.
Starting LISP → screen shortcut or by command clisp
Quitting LISP → command (exit)
Exiting debugging → command abort
LISP computations = evaluation of symbolic expressions (s-expressions)
LISP is not case sensitive and converts everything to upper case at the time of reading the input.

S-Expressions

(operation arg1 arg2 … argn)

or

atom

S-expression is short for symbolic expressions.
Operation is a function and some take fixed and some variable number of arguments.
An atom is a number or a symbol, these are elements that do not have parts.

The LISP Evaluation Rule

Look at the outermost list first, evaluate each of its arguments, and use the results as arguments to the outermost operator.

Note: the parentheses must be balanced in an s-expression or you get an error.
Note: numbers, symbol t, and symbol nil evaluate to themselves.
Tip: use new lines and indentation to make the code more readable and to keep track of parenthesis balance.

Example

(* (+ 8 3) (+ 4 (* 9 2)))
→ 242

Note: (+ 8 3) is a supplied argument to the operator *, the actual argument to the operator * is 11 which is the result of evaluating (+ 8 3).

Symbols

Symbols in LISP are the equivalent of variables.
Values are associated with symbols using the function SETQ

(setq x 5)
➔ 5
x
➔ 5

Note: Value of a symbol is the value associated (assigned to) with it in the most recent assignment.
Note: SETQ is a special function; it does not entirely follow the LISP evaluation rule since it does not evaluate the first argument given to it and takes it literally. There are only a few special functions.

LISP is very open-minded about the symbol names

(setq +8- 11)
➔ 11
+8-
➔ 11

Symbols have a multiple nature; they can be associated with a value and they can beassociated with function at the same time.

(1+ 8)
➔ 9
(setq 1+ 6)
➔ 6
(1+ 1+)
➔ 7

Note: it is a bad programming practice to use a symbol for both at once : a variable and a function.
Note: value of some symbols has a special meaning (e.x. + * numbers) and the redefinition of these is considered a bad programming practice or even an error.

LISP Data Types

LISP has a very rich set of data types: all the standard types, as well as, of example fractions and complex numbers.

(setq a 1/3)
➔ 1/3
(/ a 3)
➔ 1/9

(* a 3)
→ 1

Rational numbers: integers and fractions.
Floating-point numbers

- decimal notation (e.x. 18.07) or scientific notation (1.807E1)
- many precisions can be specified (just replace the E in the scientific notation)
    - S – short
    - F – single
    - D – double
    - L – long

LISP numeric results can be extremely large and with perfect precision without any effort:

(setq x (* (* 123 234 345 456 567 678 789 890) (* 123 234 345 456 567 678 789 890)))
→ 14940644713330123582430424044558314521600000
(* x x x x x x x x x)
→ 554234280207472736905197514687897839484169091834092988370785619858734510768295258596835515405956753490356814653251799860386022900464182422641504174195975421269845291959738176656410958822212681883799912723039083300124469795053149670785589142703121717824222576317200578816924501418053573901536541668813846120182595205884267222765812992537567853807717244673393679925710393926806509977600000000000000000000000000000000000000000000000000
→ 7

Lists

(a (b c) d) = list of 3 elements, the second is a list of 2 elements
(((x))) = list of one element which is a list of one element which is a list of one element x

Symbolic Operations

Arguments are s-expressions, operation is applied to such arguments, and the result is an s-expression. The result is the same sort of object as the arguments that are evaluated.

Note: we may need to suppress argument evaluation to pass an s-expression to a function, otherwise the s-expression would be evaluated and such result would be passed. This is dome using the (QUOTE x) function or ‘x to shorten the notation (i.e. syntactic shortcut). QUOTE is a special function – its argument is not evaluated and it is returned back.
Note: (QUOTE 6) i.e. ‘6 does not do anything is is equivalent to just 6 since numbers can always be evaluated and they evaluate to their value

(setq x (+ 3 4))
→ 7

(setq x '(+ 3 4))
➔ (+ 3 4)
Note: there is an ordinary function SET which is equivalent to SETQ. It evaluates its first argument. (SETQ X 'Y) is equivalent to (SET 'X 'Y). SET is hardly used.

(setq b 5)
➔ 5
(setq a 'b)
➔ b
(set a 6)
➔ 6
b
➔ 6

Forms are s-expressions intended for evaluation (not for manipulation)
Symbolic operations take lists apart and build them up again to produce results.

CAR function – returns the 1st element of a list i.e. return head
CDR function – returns the list without the 1st element of a given list i.e. skip head
CAR and CDR are non destructive and return a newly constructed structures.

(car '(a b c))
➔ a
(cdr '(a b c))
➔ (b c)

CAR and CDR can be composed and abbreviated

(car (cdr (cdr '(a b c))))
➔ c
(caddr '(a b c))
➔ c

Note: program code can be handled the same as data in LISP. You can manipulate code:

(car '(cdr '(a b c)))
➔ cdr

Empty List aka NIL

An empty list (), also represented equivalently as NIL, is a special symbol in LISP. It evaluates to itself by convention so it does not need quoting. You cannot change its value. It also has a special meaning in LISP (similar to NULL in Java).

‘()

➔ nil

()

➔ nil

nil

➔ nil

(cdr ‘(a))

➔ nil

Constants

A constant in LISP is:

- NIL
- Number
- Quoted s-expression

List Construction

A list can be constructed from head and tail (which must be a list) by the CONS function i.e. add the first element to a list. CONS is non destructive and returns a newly constructed structure.

(cons ‘a ‘(b c))

➔ (a b c)

(car (cons ‘a ‘(b c)))

➔ a

(cdr (cons ‘a ‘(b c)))

➔ (b c)

You can also construct a list from individual elements using the LIST function

(list ‘a ‘c ‘e ‘g)

➔ (a c e g)

Or you can append several lists into one

(append ‘(a b c) ‘(1 2 3))

➔ (a b c 1 2 3)

Note: (cons ‘(a b) ‘(c d)) produces ((a b) c d))

Note: (cons ‘a nil) produces (a)

Internal List Representation

Regular LISP structures are represented internally as binary trees (although you can build any graph with some special functions).
The nodes, also called cons cells or conses, have a right pointer and a left pointer and the terminal nodes are atoms. The node pointed by the left pointer can be obtained by using the CAR function, and the node pointed by the right pointer can be obtained by using the CDR function.

The CONS function constructs a tree node and sets the left pointer to the first element and the right pointer to the second element.
(cons ‘a ‘b) produces a cons cell that is denoted by a dotted pair (a . b), if b is nil then “. nil” can be omitted in the denotation, thus (cons ‘a nil) is denoted (a)
Every denotation of LISP list has an equivalent dotted pair denotation, but not all dotted pair denotations have a list denotation. The list notation is just for human convenience but the dotted notation better reflects the internal representation.

(a b c) ⇔ (a . (b . (c . nil)))

Dotted notation is dangerous; you can create non list objects, (a . b) is not a list unless b is nil.
CAR and CDR work on conses. You can use the CONSP function to check if an object is a cons cell.
Note: LISTP does what CONSP does plus classifies NIL as a list, it does not check if a given object is a proper list structure.

(listp ‘(a b))
➔ T
(consp ‘(a b))
➔ T
(listp ‘(a . b))
➔ T
(consp ‘(a . b))
➔ T
(listp nil)
➔ T
(consp nil)
➔ NIL

Note: NIL is a list but not cons!!!
Note: List manipulation is really a tree manipulation.

Equality

(setq x ‘(a b c))

(setq y '(a b c))
(setq z x)
(equal x y)
➔ T
(eq x y)
➔ NIL
(eq x z)
➔ T

EQUAL tests it two structures are equivalent, EQ tests if two symbols refer the same structure in memory.
Note: there is no guarantee that referencing a numeric value (ex. 5 and 5) will point to the same atom thus (eq 5 5) may sometimes not work, therefore to test if two symbols point to the same object or the same value of the same type you should use EQL function.
Note: EQ and EQL work for symbols i.e. (eq 'a 'a) → T

Destructive List Manipulation

There are functions that will manipulate the memory directly rather than build a new object and return it. They are dangerous and power full because they allow building any graph structures including circular ones.
RPLACA – substitutes for the left pointer of a cons cell
RPLACD – substitutes for the right pointer of a cons cell

(setq x '(a b c))
(setq y (cdr x))
(rplaca y 'c)
x
➔ (a c c)
(rplacd y 'd)
x
➔ (a c . d)

The destructive list manipulation functions are very powerful (you can do things that no other functions can do) and very efficient.

NCONC – glues two lists destructively
APPEND – creates a new list and by copying and glueing the given lists

Property Lists

Symbols can have a list of properties associated with them in addition to their value and the function.

(get ‘chair3 ‘color) → retrieves a property (a pointer to a property actually) COLOR of symbol CHAIR3
(setf (get ‘chair3 ‘color) ‘red) → sets the property COLOR of object CHAIR3 to RED

(get ‘chair3 ‘color)
→ NIL
(setf (get ‘chair3 ‘color) ‘red)
→ RED
(get ‘chair3 ‘color)
→ RED
(setf (get ‘chair3 ‘owner) ‘john)
→ JOHN

The properties of a symbol are stored as a list (key1 val1 key2 val2 … keyn valn), the GET function may return NIL that indicates that the property is indeed NIL or that it has not been stored yet. To avoid this confusion use some sentinel value to initiate a property and if GET returns such sentinel then you know it is set but to the “UNSPECIFIED” value.

Note: Properties were used in the past to do Knowledge Representation, where symbols represented real-world objects.
Note: Property lists are global, if you set a property for a local variable you really set it globally for the symbol used as a local variable.

Unattached Property List

You can define a list of the form (key1 val1 key2 val2 … keyn valn) and handle it just like symbol property lists.

(setq plist ‘(john ford mary kia alan bmw))
→ (john ford mary kia alan bmw)
plist
→ (john ford mary kia alan bmw)
(getf plist ‘mary)
→ kia
(setf (getf plist ‘mary) ‘gm)
→ gm
plist
→ (john ford mary gm alan bmw)

Note: SETF may change the sequence of elements but the key-value association is preserved.